FIT2099 - Assignment 1
Design Rationale

Members:

Abhishek Shrestha
Amindu Kaushal

Objective: Explain how the system will work. Describe reasoning behind design decisions

Design Principles:

1. Avoid Excessive use of literals
2. Don't repeat yourself
3. Classes should be responsible for their own properties
4. Reduce dependencies as much as possible
5. Group elements that must depend on each other inside an encapsulation boundary
6. Minimize dependencies across encapsulation boundaries
7. Declare things in the tightest possible scope
8. Delegation over inheritance
9. Fail fast principle
10. Command query principle (method can change state or return state but not both)
11. Liskov design principle

Plan: Break rationale down into separate components (features / interactions)

Items that Populate the GameMap (Tree, Dirt, Bush)

In this section, we have included objects that will be present in the world as terrain. These include the Tree and Dirt classes from the game package. We have added a new class Bush, that **inherits** Ground (similar to Tree and Dirt). This allows us to use the existing functionality of the Ground class as such tick() and can allowableActions(). Similarly, we have **inherited** the PortableItem class to create a new Fruit class, as we fruits can be picked by the player. This is an example of the Don't Repeat yourself (DRY) principle.

Though the assignment specifications hint on the Tree and Bush class having **association** with the Fruit class, we have Reduced Dependencies between these classes by deciding to store the number of Fruits each Tree or Bush object will have as integers, instead of storing a collection of Fruit objects for each Tree or Bush. Whenever other Actors interact with the Trees or Bushes to search for Fruit or when fruits fall from trees, given that the interaction is successful, we plan on instantiating a **new instance** of Fruit at the location of the Tree or Bush. Since a Tree or Bush class only needs to deal with producing fruits, we have a growFruitAction class (**extends** Action), that simply increments the number of fruits in a bush

or tree. This simply **changes the state** of the game, and nothing more, adhering to the command query principle.

TODO: growBushAction

Dinosaur Classes

In this section, we will talk about the three dinosaur classes Stegosaur, Brachiosaur and Allosaur. Since these classes would be very similar, we have made an abstract base class called Dinosaur, that in turn inherits the Actor class from the engine. This allows us to store a lot of methods and properties that three dinosaurs would share such as food level, capability (carnivorous or herbivorous), mate behavior etc. in a single abstract class, following the Don't Repeat yourself (DRY) principle. Creating three different classes instead having a of a single Dinosaur class and creating multiple types of dinosaurs objects by specifying "Stegosaur", "Brachiosaur" or "Allosaur" avoids excessive use of literals in the code, and makes the code more **structured**.

Similarly, by having a separate class for Allosaur, we can deal with actions such as EatMeatAction without involving other dinosaur classes, following the principle Declare things in the tightest possible scope, and Classes should be responsible for their own properties.

Dinosaurs and Mating

Some actions/behaviors are common to all Dinosaurs whereas some are more specific. We have have added the actions and behaviors common to all dinosaurs (eg. Mate Behavior and MateAction) to the base class Dinosaur. This allows us to follow Liskov's principle, as all subclass objects will be able to replace the superclass object without change in functionality. Speaking of behaviors, we have added the Behavior interface to the base class Dinosaur and want to store it as a collection (eg. ArrayList). So all dinosaurs can have a list of behaviors. FollowBehavior **implements** the Behavior **interface**, which gives dinosaurs the ability to follow a target, which is done by returning a MoveActorAction. Instead of creating a new followBehavior for finding a mate, we decided to **extend** the FollowBehavior class, as it already has most of the functionality we need. Here, we can set the target to be the dinosaur of same specie but opposite sex. Depending on the distance between the target dinosaur and the actor dinosaur, this behavior calls the MoveActorAction (from super) or MateAction class, which extends Action from the engine. If a mate is successful, this same behavior can then call the LayEggAction (**extends** the action class), which would allow the female dinosaur to lay an egg in the location that she is in (by querying the current location via the GameMap and Location classes). Calling the correct Action class would just be implementing some conditionals.

Brachiosaur and Bush

To implement Brachiosaur destroying a bush, we have created a DestroyBushAction class that **extends** the Action class. It has a target bush object and the brachiosaur that is going to destroy the bush. By confirming that the bush and Brachiosaur are at the same location (querying from the Location and GameMap classes), we create a new dirt object at the same location, successfully destroying the bush. By doing so, we have only included essential classes, and have managed to declare things in the tightest possible scope.

Dead Dinosaurs and Corpse

We have created an **abstract** Corpse class by **inheriting** the given PortableItem class, as a corpse can be picked up by the player. Similarly, we need to distinguish between corpses as eating a Brachiosaur corpse would completely fill an Allosaur but the corpse of any other dinosaur would not. Thus, we have used the Corpse class as a **base class** for inheriting the StegosaurCorpse, BrachiosaurCorpse and AllosaurCorpse classes. This is again another use of the Don't Repeat Yourself (DRY) principle. Same as with the Dinosaur classes, having separate Corpse classes allows us to avoid excessive use of literals in code.

Seek Food for carnivore and herbivore

For the herbivores, we have added a SeekFruitBehavior class to the list of behaviors for that dinosaur. For the carnivore, we have added a SeekMeatBehavior class to the list of behaviors for that dinosaur. Both these new classes **implement** the Behavior interface, instead of extending an existing class that implements it because implementing interfaces allows code to be more **extensible**. This follows the delegation over inheritance and Don't repeat yourself (DRY) principles. Both behavior classes have targets: Fruit for SeekFruitBehavior and (Stegosaur, Corpse and Egg) for SeekMeatBehavior. The SeekFruitBehavior or SeekMeatBehavior class calls the MoveActorAction class (depends on Exits, Location and GameMap class) if the dinosaur is away from the target. However, if the dinosaur and the target are in the same location (which is queried from the Location and GameMap) class, then it calls the EatFruitAction or EatMeatAction, which would remove the fruit from the target and increment that dinosaur's foodlevel. If the target is a stegosaur for the SeekMeatAction, then it would call the AttackAction. As these actions would simply change the state of the game and not query anything, the code follows the command query principle.

Vending Machine and functionality

We created a Vending machine class that would store all the given items and would have an infinite number of each item. Each Item the vending machine stores extends the PortableItem class, as the player should be able to pick up all items in the vending machine. Since players should be able to purchase a LazerGun, we have created a LazerGun class by **extending** the WeaponItem class from the engine. WeaponItems are portable by default.

The BuyItemAction extends the Action base class from the engine. All inheritances are examples of Don't repeat yourself (DRY). The BuyItemAction (see interaction diagram) allows the player to purchase an item (possibly shown by a menu) given there are enough ecoPoints. If so, that item will be instantiated and added to the player's inventory.

Players interaction with Dinosaurs

Since players should be able to feed dinosaurs, we have created a FeedAction (see class and interaction diagram), which **extends** the Action class. The player would be the initiator (property) for the FeedAction class. This class would check if the player's inventory has a suitable food item for the dinosaur the player wants to feed. This is accessed through the getInventory method of the player. However, since different dinosaurs eat different types of items, we have **extended** the FeedAction class into FeedBrachiosaur, FeedStegosaur and FeedAllosaur classes to avoid excessive use of literals and to reuse code (following DRY). This also avoids unnecessary conditionals and adds **separation of concerns** for each of the dinoFeedAction classes.

Players also have to control the Stegosaur population. Thus, we created a ControlPopulation class, which **inherits** the AttackAction class. If the player and the target (only Stegosaur uptil now) are close enough, the player can attack the Stegosaur by overriding the execute() method of the AttackAction class. Here, we have made a new ControlPopulation class instead of directly using the provided AttackAction class in order to make our code more **extensible**. Later, adding the functionality to attack other dinosaurs would be simple, by just replacing the target from Stegosaur to Dinosaur. Changing the target to the base class Dinosaur would then follow Liskov's design principle, as any dinosaur can be attacked by the player.